Practical Static Analysis for NASA

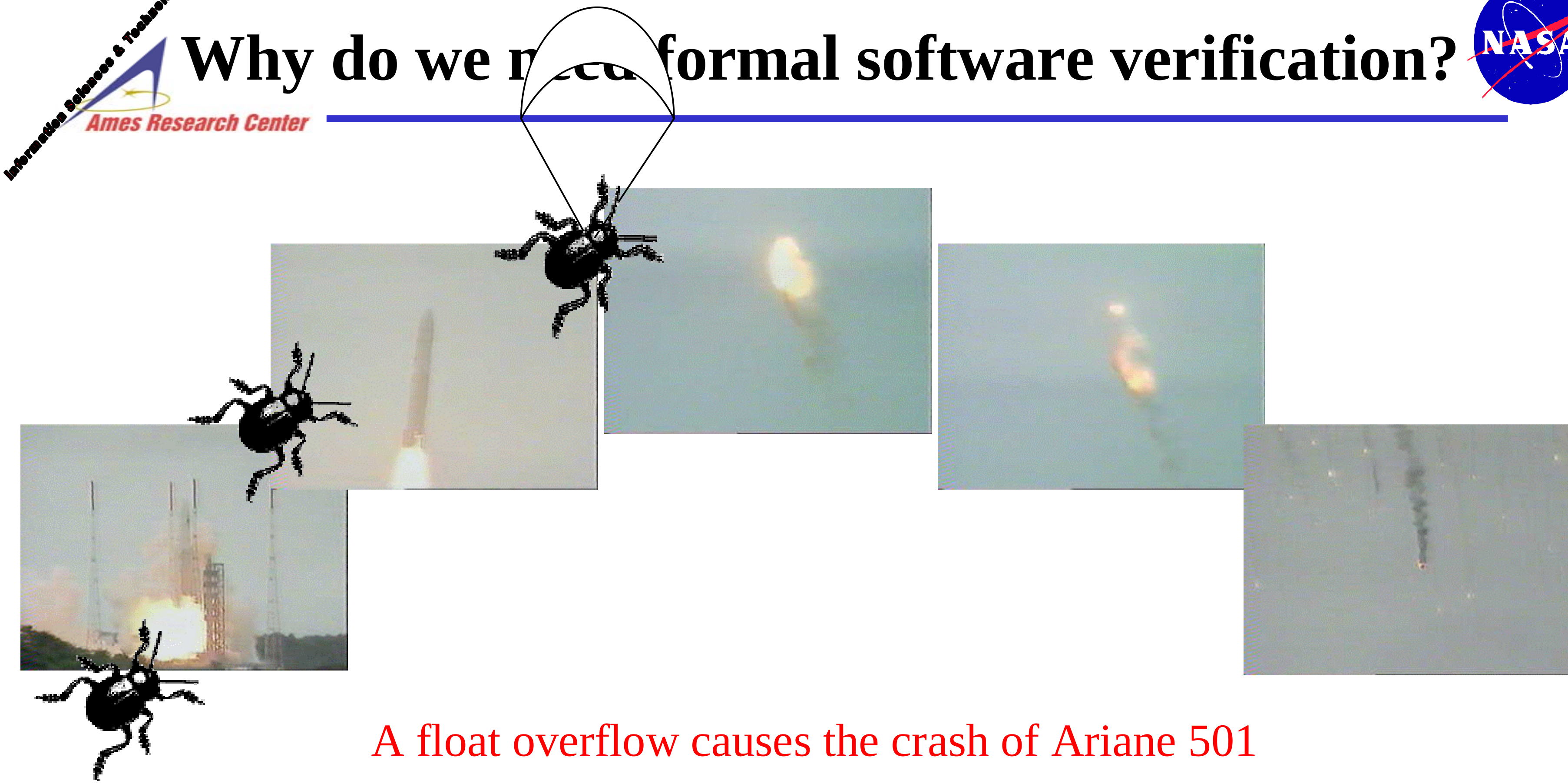
Guillaume Brat
and
Arnaud Venet

Kestrel Technology
NASA Ames Research Center

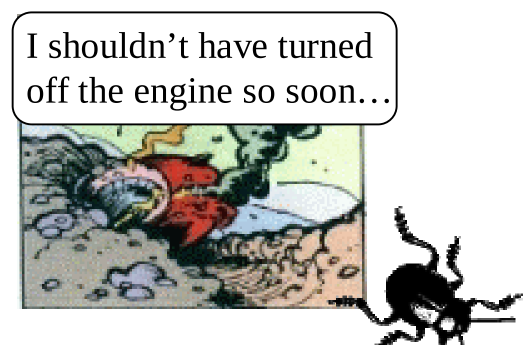
Outline

- Motivation
- Static analysis
  - Quick overview
  - Targeted error classes
- Research goal elicitation
  - The MPF experience
  - Research gaps
- The present
  - C Global Surveyor
  - Status
- Future work
  - Mission impact
  - MDS

Why do we need formal software verification?



A float overflow causes the crash of Ariane 501



A badly initialized variable caused Mars Polar Lander to crash on Mars

Static Analysis

all possible values
(and more) are computed

the analysis is done
without executing the program

Static analysis offers compile-time techniques for predicting safe and computable approximations to the set of values arising dynamically at run-time when executing the program

We use abstract interpretation techniques
to extract a safe system of semantic equations
which can be resolved using lattice theory techniques
to obtain numerical invariants for each program point

Covered Defect Classes

- Static analysis is well-suited for catching runtime errors, e.g.:
  - Array-out-bound accesses
  - Un-initialized variables/pointers
  - Overflow/Underflow
  - Invalid arithmetic operations
- Defect classes for Deep Space One:
  - Concurrency: race conditions, deadlocks
  - Misuse: array out-of-bound, pointer mis-assignments
  - Initialization: no value, incorrect value
  - Assignment: wrong value, type mismatch
  - Computation: wrong equation
  - Undefined Ops: FP errors (tan(90)), arithmetic (division by zero)
  - Omission: case/switch clauses without defaults
  - Scoping Confusion: global/local, static/dynamic
  - Argument Mismatches: missing args, too many args, wrong types, uninitialized args
  - Finiteness: underflow, overflow

Research Process

precision scalability

Identification of
technical gaps

usability

Identification of
commercial tools

PolySpace
C-verifier

MPF DS1

Experiments on
real NASA code

ISS K9

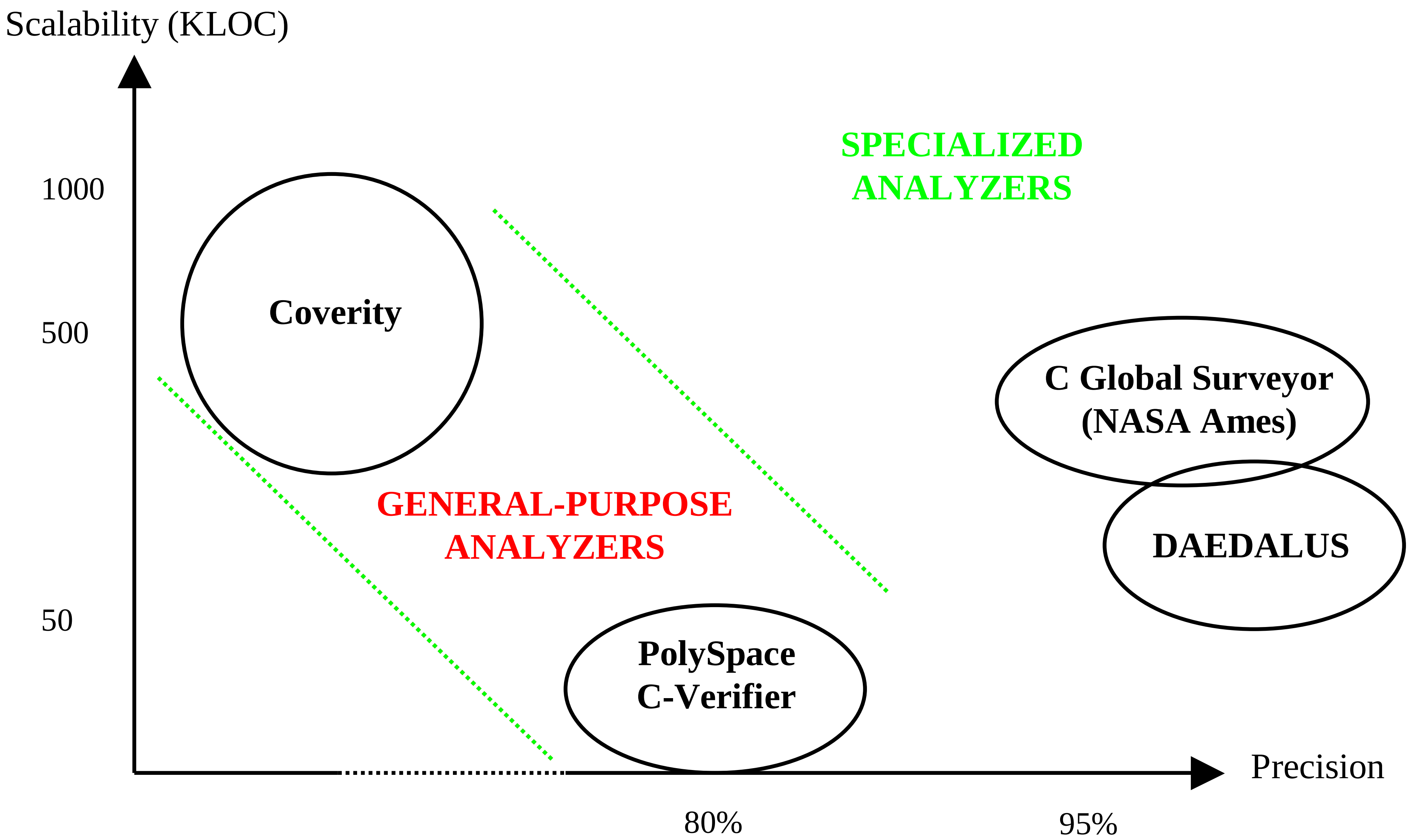
Implementation of
research prototype

CGS

PolySpace applied to Mars PathFinder

- Analyzed 3 modules (~20KLoc each) of mature C code for runtime errors (RTEs)
- Performed the analysis at level of system integration
  - MPF testing was really done at the validation phase
- 80 % Selectivity
  - 80% checks have been classified (correct or incorrect) with certainty
  - 20% warnings: need to be covered by conventional testing
- Found 2 certain errors in 30 minutes
  - But, average run is 12 hours
  - Average time spent manually analyzing RTE is 0.5 hours
- ACS module was fairly mature:
  - Only 1 red check (NIV) in 25KLocs with 3 threads
  - Not critical, but prevented optimization code to execute
  - Error is similar to the one that caused Mars Polar Lander’s crash

Practical Static Analysis



Design Factors

PolySpace Limitations

- Precision:
  - Array cells merge into one
- Scalability: limited by
  - Size (< 20KLocs)
  - Pointer analysis
  - Multithread combinatorics
- Result interpretation
- Usability

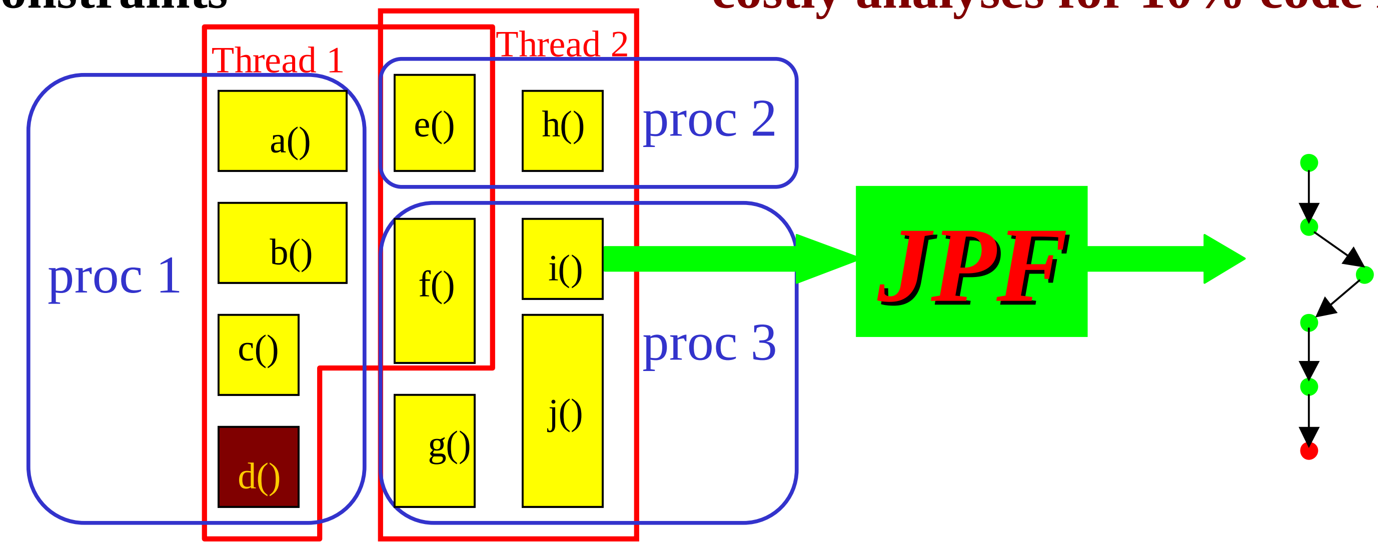
MPF Legacy Coding Practice

- Base data structure: matrix
- Pointers are mainly used
  - to iterate over matrix elements
  - in complex loop structures
- Mostly static data
  - Marginal use of dynamically allocated structures
- Several threads of execution

C Global Surveyor

Specialized pointer analysis
precise for top-level pointers
thread sensitive
Supplement pointer info
 with index range constraints

Incremental refinement of analyses
build analyses on top of each other
simple analyses for 90% of code
complex analyses refines simpler ones
costly analyses for 10% code left



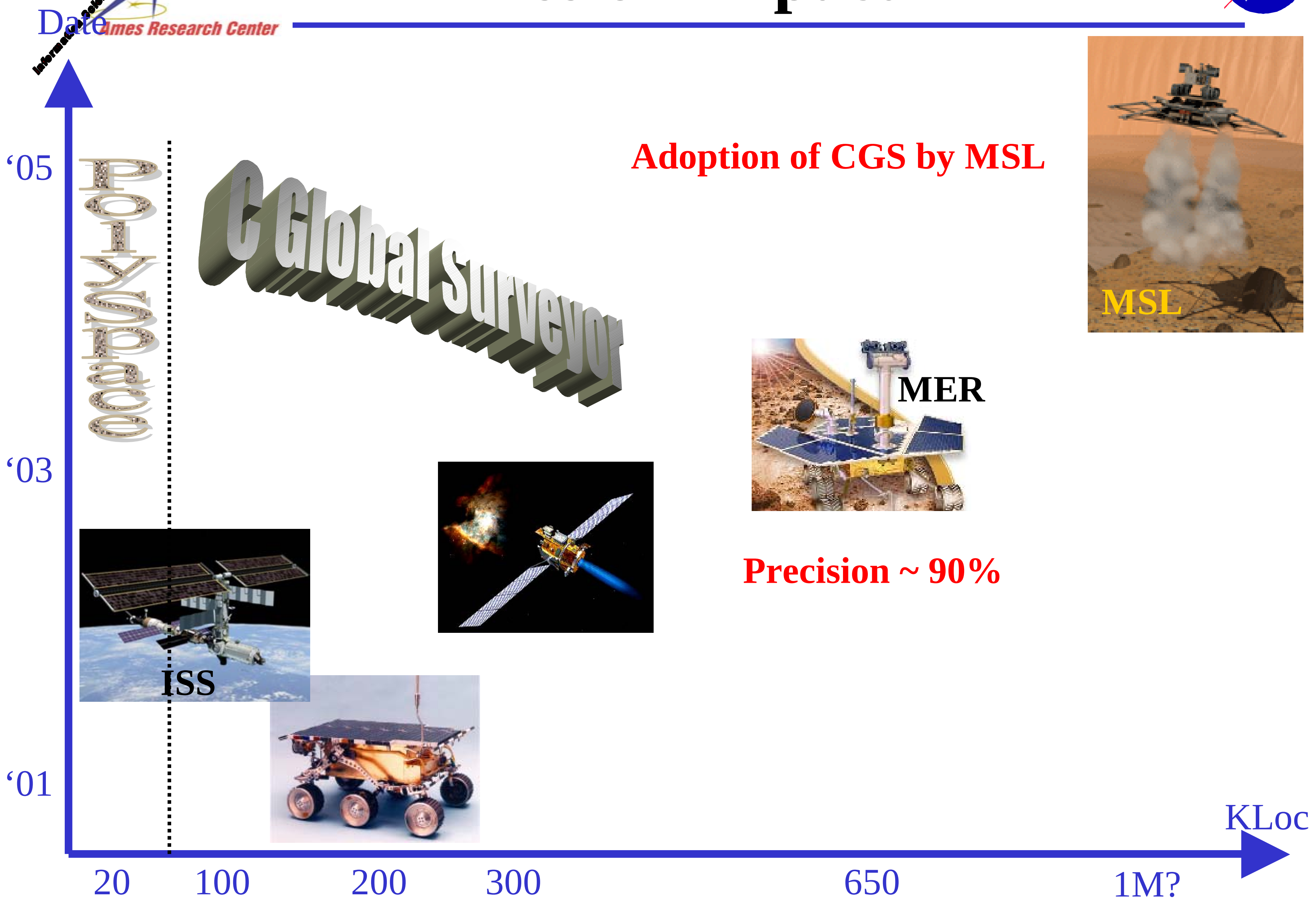
granularity of algorithms is function
context passing:
 low overhead w.r.t. computation time
Distributed abstract interpretation

use JPF to generate scenarios
to illustrate certain errors
and to filter false positives
Smart result interpretation

CGS Status

- Prototype is fully implemented
  - Surface pointer analysis
  - Array-bound checking
- Current performance on dual 2.2 GHz processor with 2 GB memory:
  - 45 minutes for MPF (132 KLoc w/o *.h)
  - 1 hour 45 minutes for DS1 (275 KLoc w/o *.h)
- Currently under implementation:
  - Precise pointer analysis

Mission Impact



An MDS Approach

- Goal: building a static analyzer for MDS using specialization
- The idea is to perform V&V at two levels
  - Framework level
    - Prove very strong semantic properties about the MDS framework
  - Adaptation level
    - Verify that the code using the MDS framework does the right thing
    - Brings static analysis up one level of abstraction towards the system level
- Concrete steps using two examples:
  - Exception safety checking
    - E.g., release locks that were acquired
  - Safety checking at pattern level
    - E.g., reference-counted smart pointer

Conclusions

- Using static analysis to catch runtime errors
- Ran experiments with commercial tools on real NASA software systems (< 275 KLoc)
- Identified scalability and precision problems
- Implemented a scalable static analyzers specialized for MPF-based NASA software
- Will use the same philosophy to design a static analyzer for MDS applications (MSL mission)